Conceito Gerais:

- → Definição de Sistema Operativo;
  - → Para que são otimizados:
    - Em servidores;
    - Em *desktop/laptop*;
    - Em *tablets/smartphones*;
    - Em sistemas embutidos;
- → Breve história do sistema UNIX;
  - → Caracteríticas dos sistemas UNIX;
  - → Sistemas mais comuns e a sua linhagem:
    - LINUX;
    - MAC OS X;
    - Windows;
    - iOS;
    - Android;
- → Funções básicas de um SO;
- → Arquitetura:
  - *Kernel*;
  - Serviços típicos;
  - *System Calls*;
- → Multi-programação e Multi-tarefa;
- → BIOS:
  - O que é;
  - Para que serve;
- → Bootstrap no UNIX;
- → O que são *interrupts* e *traps* (exemplos);
- → Interação com dispositivos:
  - Controladores dos dispositivos;
  - *Device drivers*;
  - DMA (para dispositivos com elevada largura de banda);
- → Proteção do sistema:
  - Modos *kernel* e *user*;
  - *System Calls* (como *traps*);
  - Implementação;

UNIX

O Unix é um sistema operativo criado no início dos anos 70, principalmente por *Dennis Ritchie* e *Ken Thompson*. As suas principais características técnicas são sua portabilidade, sua capacidade de multi-utilizador e de multitarefa, eficiência, alta segurança e o bom desempenho em tarefas de rede. Existem vários sistemas operativos conhecidos como derivados do Unix, entre eles, o ***Linux***, ***MacOS***, ***Android*** e o ***iOS*** (derivado do ***Mac OS X***).

Linux, foi criando no início dos anos 90 por *Linus Torvalds* quando o núcleo desenhado por *Linus* foi distribuído juntamente com as ferramentas ***GNU***. O projeto GNU criou a licença de software GPL, que garante as liberdades de uso, modificação e colaboração a respeito do software. Desta forma, as ferramentas do GNU são desenvolvidos, revistas, melhoradas e adaptadas para milhares de utilizadores e centenas de empresas ao redor do mundo. Além disso, este conjunto está disponível gratuitamente e seu código de fonte é aberto.

MacOS foi desenvolvido pela *Apple Inc.* A sua versão clássica foi desenvolvida entre 1984 até 2001 e depois criou-se o modelo mais desenvolvido que ainda é usado hoje em dia. É o segundo sistema operativo mais usado atualmente, estando atrás do ***Microsoft Windows***, desenvolvido pela *Microsoft* em 1985.

Android é um sistema operativo móvel baseado em uma versão modificada do *kernel Linux* e outro software de código aberto, projetado principalmente para *touchscreen,* desenvolvido por um conjunto de firmas chamado de *Open Handset Alliance* em 2007 e comercializado pela *Google*.

i***OS as*** é um sistema operativo móvel criado e desenvolvido pela *Apple Inc* e lançado em 2007, exclusivamente para o seu *hardware*. É o segundo sistema operativo móvel mais instalado, depois do *Android*.

Sistemas Operativos

Definição do livro: Um sistema operativo é o único programa em execução o tempo todo no computador – geralmente chamado de *kernel*.

Definição da aula: Um sistema operativo é um *software* que providencia os recursos de *hardware* de um computador necessários à execução das aplicações dos seus utilizadores.

- Atua como intermediário entre o utilizador e o *hardware*
- Providencia o básico para as aplicações
- Alguns são desenhados para tornar o sistema do computador conveniente e de fácil utilização, outros são desenhados para usar o *hardware* com maior eficiência e outros para combinar os dois.

Os sistemas operativos são desenhados para corresponder da melhor forma aos dispositivos em que estão.

- **Servidores**: Projetado para maximizar a utilização de recursos e garantir que todo o tempo de CPU, memória e I/O disponíveis sejam usados de maneira eficiente e justa entre todos os utilizadores -***"Keep all users happy!"***
- ***Desktop/Laptop:*** Otimizados para a experiência de um único utilizador em vez dos requisitos de vários utilizadores. Projetado para facilitar a utilização com alguma atenção ao desempenho e quase nenhuma para a utilização dos recursos;
- ***Tablets/Smartphone:*** Otimizados para usabilidade e minimização do gasto da bateria. Projetado para facilidade de uso, com atenção especial à utilização dos recursos;

- ***Embedded Computers:*** A maioria dos computadores embutidos, em dispositivos domésticos e automóveis, têm pouca ou nenhuma interface para o utilizador. Projetado para ser executado sem/mínima intervenção do utilizador;

Um sistema de um computador pode ser dividido em quatro partes:

- ***Hardware*** – recursos básicos do sistema (CPU, memória, dispositivos I/O…);
- **Sistema operativo** – controla e coordena o uso do *hardware* com as várias aplicações e utilizadores;
- **Programas –** definem a maneira como os recursos do sistema são usados para resolver as necessidades dos usuários (processadores de texto, navegadores *web*, sistemas de base de dados, jogos, compiladores, ...)
- **Utilizadores –** pessoas, outros programas/computadores;

O sistema operativo moderno geralmente inclui os seguintes componentes principais:

1) Gestão de **processos**
   - Criar, suspender, retomar e encerrar processos (usuário/sistema);
   - Fornecer mecanismos para comunicação de processos;
   - Fornecer mecanismos para sincronização de processos;
   - Fornecer mecanismo para tratamento de impasses (*deadlock)*
2) Gestão de **memória**;
   - Alocação e desalocação de espaço de memória conforme necessário;
   - Rastreamento de quais partes da memória estão a ser usadas no momento e por quem;
   - Decidir quais processos/dados devem ser movidos para dentro e para fora da memória e quando;
3) Gestão de **armazenamento**;
   - Fornecer visão uniforme e lógica do armazenamento de informações;
   - Suporte para criar, excluir e manipular ficheiros e diretórios;
   - Políticas de controlo de acesso para determinar quem pode acessar o quê;
   - Mapeamento/*Backup* de ficheiros em dispositivos de armazenamento secundário não volátil;
4) Gestão de dispositivos de **I/O**;
   - Oculta peculiaridades dos dispositivos de *hardware* do usuário;
   - Responsável pela gestão de memória de I/O;

O ***kernel*** é o centro essencial de um sistema operativo de computador. É o núcleo que fornece serviços básicos para todas as outras partes do sistema operativo. É a camada principal entre o sistema operativo e o *hardware* e ajuda na gestão de processos e memória, sistemas de ficheiros, controle de dispositivos e rede.

Os serviços prestados diferem de um sistema operativo para outro, mas podemos identificar classes comuns:

- Interfaces de usuário – para permitir operação e controle eficazes do sistema **(*User Interface*)**;
- Execução do programa – para carregar um programa na memória e executá-lo **(*Program Execution*)**;
- Operações de I/O – para fornecer um meio de realizar operações de I/O **(*I/O Operations*)**;

- Sistemas de ficheiros – para permitir a manipulação eficaz de ficheiros e diretórios **(*File Systems*)**;
- Comunicações – para permitir a troca de informações entre processos no mesmo computador ou entre computadores em uma rede **(*Communications*)**;
- Deteção de erros – estar constantemente ciente de possíveis erros que podem ocorrer no *hardware* da CPU/memória, nos dispositivos de I/O ou nos programas do usuário, a fim de tomar as medidas apropriadas para garantir uma computação correta e consistente **(*Error Detection*)**;

Outro conjunto de serviços existe não para ajudar o utilizador, mas sim para garantindo o funcionamento eficiente do próprio sistema:

- Alocação de recursos – quando vários processos estão sendo executados simultaneamente, os recursos disponíveis (como ciclos de CPU, memória principal, armazenamento de ficheiros, dispositivos de I/O) devem ser alocados de forma eficiente para cada um deles **(*Resource allocation*)**;
- Contabilidade – para acompanhar quais utilizadores usam quanto e quais tipos de recursos do computador **(*Accounting*)**;
- Proteção e segurança – para evitar que processos simultâneos interfiram uns com os outros ou com o próprio sistema operativo e para proteger o sistema de terceiros **(*Protection and Security*)**;

As ***system calls*** fornecem uma interface para os serviços do sistema operativo. São acessadas a partir de uma *high-level application program interface (API)*. Podem ser agrupadas em seis categorias:

- Controlo de processo (*Process Control);*
- Manipulação de ficheiros (*File Manipulation*);
- Manipulação de dispositivos (*Device Manipulation*);
- Manutenção de informações (*Information Maintenance*);
- Comunicação (*Communication*);
- Proteção (*Protection*);

Um dos aspetos mais importantes dos sistemas operacionais é a capacidade de ter **vários programas em execução.** A **multiprogramação** aumenta a utilização da CPU organizando os processos para que a CPU possa sempre executar um trabalho.

(1) O sistema operativo começa a executar um processo por meio do agendamento de processos (*job scheduling)*;
(2) Eventualmente, o trabalho pode ter que esperar por alguma tarefa, como uma operação de I/O;
(3) Em um sistema não multiprogramado, a CPU ficaria inativa;
(4) Em um sistema multiprogramado, o sistema operativo alterna para outro trabalho. Quando esse trabalho precisa esperar, a CPU muda novamente para outro trabalho e assim por diante. Eventualmente, o primeiro trabalho termina de esperar e recupera a CPU;
(5) Enquanto pelo menos um trabalho precisa ser executado, a CPU nunca fica inativa;

A **multiprogramação** aumenta a utilização da CPU, mas não fornece necessariamente a interação do utilizador com o sistema do computador. ***Multitasking*** (multi-tarefa) é uma extensão da **multiprogramação** que aumenta o tempo de resposta em que a CPU alterna tarefas com tanta frequência que os utilizadores podem interagir com cada tarefa enquanto ela está em execução.

BIOS (Basic Input/Output System) é o primeiro programa responsável pela inicialização do sistema do computador (*bootstrap program)*, é carregada na inicialização ou no *reboot*.

- Armazenada em memória de leitura apenas (ROM, *read-only memory*);
- Inicializa todos os aspetos do sistema, desde dos registros da CPU para controladores de dispositivos até conteúdos da memória;
- Carrega o *kernel* do sistema operativo e inicia a sua execução;

Assim que o *kernel* é carregado, pode começar a fornecer os serviços disponíveis aos utilizadores. Alguns serviços são fornecidos fora do *kernel*, por processos do sistema que são carregados no momento da inicialização (no UNIX, o primeiro processo do sistema é o processo *init* que inicia muitos outros processos do sistema). Assim que esta fase é concluída, o sistema é totalmente inicializado e começa a aguardar a ocorrência de algum evento.

A ocorrência de um evento geralmente é sinalizado por uma interrupção (***interrupt***) do *hardware* ou do *software.* O *hardware* pode acionar uma interrupção a qualquer momento enviando um sinal para a CPU, para comunicar que ele precisa da atenção do sistema operativo. O *software* pode acionar uma interrupção executando uma operação especial chamada de *System Call.*

Quando a CPU é interrompida, suspende a atividade corrente, guardando o estado em que está, e transfere , de imediato, a execução para uma função fixa chamada de *interrupt handler* (manipulador de interrupção) para lidar com o evento. Após a conclusão, a CPU retoma a computação do processo que tinha interrompido.

Traps são levantadas pelo programa do utilizador para invocar uma funcionalidade do sistema operativo. Suponha que o programa do utilizador exija a impressão de algo na tela. Ele invocaria uma ***trap*** e o sistema operativo executaria essa instrução. São usadas principalmente para implementar ***system calls***. Um ***interrupt*** é gerado por um dispositivo de *hardware*, as interrupções são assíncronas, isto é, podem ocorrer a qualquer momento. Dispositivos como teclados são conectados ao processador através do pino de interrupção. Quando uma tecla é pressionada, ela gera uma interrupção. O processador mudará do processo atualmente em execução para um *interrupt handler*. Nesse cenário, o manipulador de interrupção do teclado é chamado. Depois de completar a rotina de tratamento de interrupção, o processador volta para o programa original que estava sendo executado.

Os sistemas do computador consistem em vários **controladores de dispositivos** (componentes de *hardware*) conectados por meio de um *bus – sistema de comunicação que transfere dados entre componentes dentro de um computador ou entre computadores*. Para se comunicar com cada controlador de dispositivo, os sistemas operativos exigem um ***driver* de dispositivo (*Device drivers*)** específico (componente de software). ***Device drivers*** ajudam o *kernel* a executar ações. São pedaços de código que correspondem a cada dispositivo e são executados quando os dispositivos se conectam ao sistema operativo ou ao *hardware*. Ajudam a fechar o espaço entre as aplicações e o *hardware.* Para garantir a funcionalidade correta, o *kernel* deve ter um ***device driver*** embutido para cada periférico presente no sistema.

Para iniciar uma operação de I/O, o **driver de dispositivo** carrega os registros apropriados no **controlador de dispositivo.**

(1) O controlador, por sua vez, examina o conteúdo desses registros para determinar qual ação tomar (por exemplo, ler um caractere do teclado).
(2) O controlador então inicia a transferência de dados do dispositivo para o *buffer* local.

(3) Uma vez que a operação de I/O foi concluída, o **controlador do dispositivo** informa o ***driver* do dispositivo** por meio de uma **interrupção.** O ***driver* de dispositivo** retorna o controlo ao sistema operativo.

Esta forma de I/O controlada por *interrupt* é boa para transmitir pequenas quantidades de dados, mas pode haver sobrecarga se os dados forem em massa. Para dispositivos de I/O de alta velocidade, capazes de transmitir informações em velocidades próximas às da memória, esse problema é resolvido usando o **acesso direto à memória (DMA – *Direct Memory Access*)**, exemplos, *graphics cards, network cards, disk drive controllers…* O **controlador de dispositivo** transfere blocos de dados de seu próprio armazenamento de *buffer* diretamente para a memória principal, sem intervenção do **driver de dispositivo** (CPU).

Um sistema operativo projetado adequadamente deve garantir que um programa incorreto ou mal-intencionado não possa fazer com que outros programas sejam executados incorretamente. A abordagem adotada pela maioria dos sistemas de computador é fornecer suporte de hardware que nos permita diferenciar entre, pelo menos, dois modos separados de operação:

- ***User Mode;***
- ***Kernel Mode;***

A operação de modo duplo permite que o sistema operativo proteja a si mesmo e a outros componentes do sistema.

Um ***mode bit*** vindo do *hardware* indica o modo atual:

- Permite distinguir quando o sistema está a correr em *user code* ou em *kernel code;*
- Algumas instruções, designadas como instruções privilegiadas, são executáveis apenas no modo kernel (instruções para controle de I/O, gestão do *timer*, gestão de interrupção, …);
- Interrupções ou *system calls* alteram o modo para o *kernel*, o retornar das interrupções ou *system calls* redefinem-no para o modo *user;*

Normalmente, um número é associado a cada *system call* e a *system call interface* mantém uma tabela indexada de acordo com esses números. A *system call interface* invoca a *system call* pretendida no *kernel* do sistema operativo e retorna o status da *system call* e quaisquer valores de retorno.

Programas e Processos

→ Programa: ficheiro binário executável;
- → Estrutura;
- → Como é obtido por compilação;

→ Processo e espaço de endereçamento
- → Segmentos e suas funções:
  - → *.text*;
  - → *.data*;
  - → *.bss*;
  - → *Heap;*
  - → *Stack;*

→ *Process Control Block (PCB);*
- → O que é;
- → Que informação contém;

→ Ciclo de vida de um processo;

→ Processo "*scheduler*";

→ "*Context Switch*";

→ Criação, gestão e terminação de processos;

→ Processos órfãos e *zombies;*

→ *Inter Process Communication* (*IPC*);

→ *API* de memória partilhada do UNIX/*System V;*

→ *API POSIX* de memória partilhada (*mmap*);

→ *Pipes;*

→ *Sockets UNIX* e *INET*;

→ Modelo cliente-servidor;

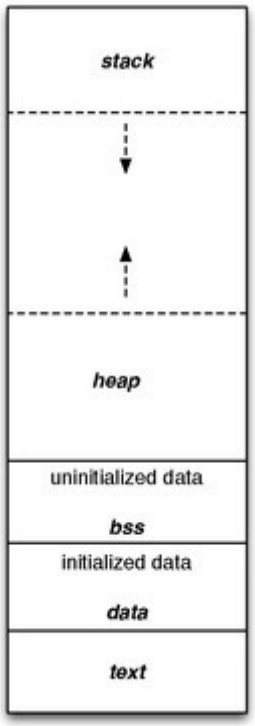
Cada ficheiro executável (**programa**) é composto por um *header*, descrevendo os segmentos de memória, seguido pelos dados do ficheiro. Os segmentos contêm informações necessárias para a execução do ficheiro durante a execução do mesmo, assim como, dados para vinculações (*linkers*) e de realocação. É produzido por compilação e ligação (*compiling and linking)* numa etapa.

Um **processo** é a unidade de trabalho na maioria dos sistemas e pode ser pensado como um programa em execução. Para cumprir o trabalho, um processo precisa de recursos, por exemplo, tempo de CPU, memória, ficheiros, etc.

Um **processo** possui várias partes:

- Secção de texto – contém o código do programa (***Text section***);
- Secção dos dados – contém as variáveis globais (***Data section***);
- *Heap* – contém memória dinamicamente alocada durante o tempo de execução do processo (***Heap***);
- Stack (Pilha) – contém dados temporários (como parâmetros de funções, endereços de retorno e variáveis locais) (***Stack***);
- Bloco de controlo do processo (PCB) – que inclui o contador do programa e os registradores da CPU (**Process Control Block**);

As partes de texto, dados, heap e pilha formam o espaço de endereço de memória do processo.



O sistema operativo representa cada processo por um ***Process Control Block (PCB)***, bloco de controlo de processo, que contém informações associadas ao processo:

- **Estado do processo** e **identificação do processo**;
- **Contador do programa**, a próxima instrução a ser executada;
- **Registradores da CPU**, **registradores de índice**, ***stack pointers***, etc;
- **Informações de agendamento**, como prioridades, *scheduling queue pointers,* etc;
- **Informações da memória**, a memória alocada para o processo;
- **Informações de contabilidade**, como CPU usada, tempo de relógio decorrido desde o início, limites de tempo, etc;
- **Informações de I/O**, como dispositivos de I/O alocados, lista de ficheiros abertos, etc.

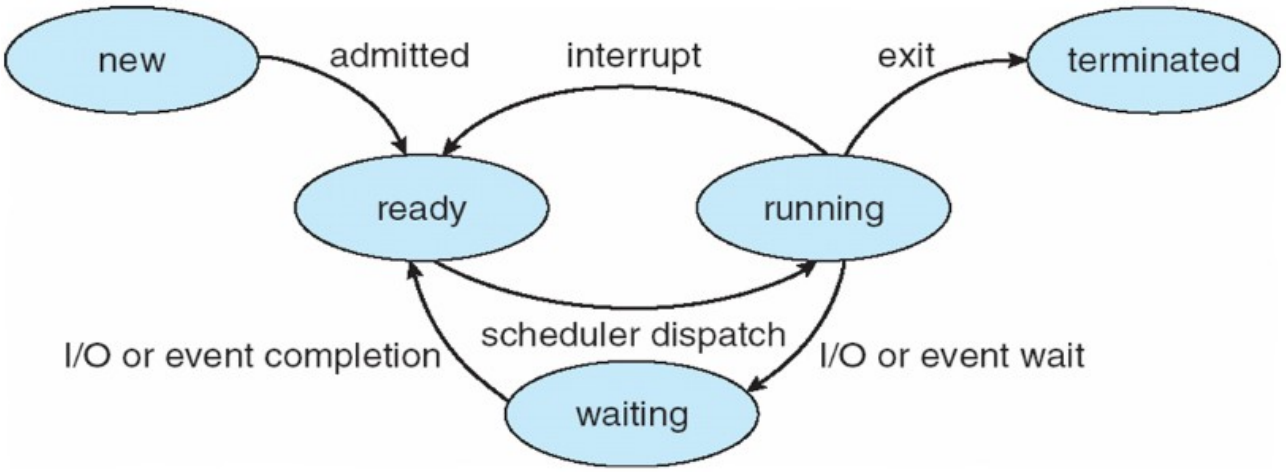
Um **programa** é uma entidade passiva (geralmente chamado de ficheiro executável) que contém uma lista bem definida de instruções (algoritmo). Um **processo** é uma entidade ativa correspondente a uma sequência de execução de um programa. Um **programa** torna-se num processo quando é carregado na memória.

- Um programa pode ser vários processos;
- Dois processos associados ao mesmo programa são considerados duas sequências de execução separadas;

À medida que um **processo** é executado, ele muda de estado de acordo com a sua atividade atual. Um **processo** pode estar em um dos seguintes estados:

- **Novo**, o processo está a ser criado;
- **Em execução**, as instruções estão a ser executadas;
- **Aguardando/Bloqueado**, o processo está a aguardar a ocorrência de algum evento;
- **Pronto**, o processo está a aguardar para ser atribuído a um processador;
- **Terminado**, o processo terminou a execução;

Muitos processos podem estar **prontos** ou **em espera,** mas apenas um processo pode ser executado em um processador de cada vez .



Process scheduler é a atividade do gestor de processos que trata da remoção do processo em execução da CPU e da seleção de outro processo com base em uma determinada estratégia. É uma parte essencial de um sistema operativo de multiprogramação.

Context Switch é a tarefa de alternar a CPU para outro processo.

- *Context Switch* requer salvar o contexto do processo atual no seu ***PCB***

O tempo do ***context switch*** é pura sobrecarga, porque o sistema não faz nenhum trabalho útil durante a troca. A velocidade de comutação depende da velocidade da memória, do número de registros que devem ser copiados e da existência de instruções especiais

- Quanto mais complexo for o SO e o PCB, maior será o ***context switch***

Durante a execução, um processo pode criar vários novos processos.

- O processo criador é chamado de ***parent process*** e os novos processos são chamados de filhos desse processo;
- Cada processo novo pode criar outros processos, formando uma árvore de processos;

A maioria dos sistemas operativos identifica os processos de acordo com um identificador de processo exclusivo (ou ***pid***), que normalmente é um número inteiro.

- O ***pid*** fornece um valor único para cada processo no sistema e pode ser usado como um índice para acessar vários atributos de um processo dentro do *kernel*

➢ Alternativas de compartilhamento de recursos:
  - Pais e filhos compartilham todos os recursos;
  - Os filhos compartilham um subconjunto dos recursos dos pais;
  - Pai e filho não compartilham recursos;

➢ Alternativas de execução:
  - Pai e filhos são executados simultaneamente;
  - O pai espera até que alguns ou todos os filhos tenham terminado;

➢ Alternativas de espaço de endereço:

- Pai e filho são duplicados (o filho começa com o programa e os dados do pai);
- O processo filho tem um novo programa carregado nele;

O que é preciso para criar um processo?

- Atribuir um identificador de processo e armazenar o novo PCB – Barato;
- Copiar o estado de I/O do pai (dispositivos de I/O alocados, lista de ficheiro abertos,…) – Meio caro;
- Configurar novas tabelas de memória para o espaço de endereço – Mais caro;
- Copiar dados do processo pai – Muito caro / Mais barato com copy-on-write;

No **UNIX/Linux,** um processo é criado pelo método *fork()* do *system call.*

- O novo processo (filho) consiste em um cópia do espaço de endereço do processo original (pai);
- O código de retorno para *fork()* é zero para o filho e o identificador do processo (***pid***) do filho é retornado para o pai (diferente de zero);
- Ambos os processos continuam a execução concorrentemente na instrução após o *fork()*;
- Esse mecanismo permite que o processo pai se comunique facilmente com os seus filhos;

Normalmente, após um *fork()*, um dos dois processos (pai ou filho) invoca a *system call exec()* – substitui o espaço de memória do processo (texto, dados, *heap*, partes da *stack*) por um novo programa do disco e começa a executar o novo programa na sua função principal, destruindo a imagem do processo anterior.

- No entanto, como nenhum processo novo é criado, o processo de chamada mantém o seu contexto (PCB);
- Dessa forma, pais e filhos podem seguir caminhos separados;

Um processo encerra a execução quando invoca explicitamente a *system call exit()* ou quando executa a última instrução.

- Todos os recursos do processo – incluindo memória física e virtual, ficheiros abertos e *buffers* de I/O – são então desalocados pelo sistema operativo;
- Um valor de *status* de saída (número inteiro) é disponibilizado para o pai por meio da *system call wait()*;

Um processo também pode encerrar a execução por meio do seu pai:

- A criança excedeu os recursos alocados;
- A tarefa atribuída à criança não é mais necessária;
- O pai está saindo e o sistema operativo não permite que o filho continue se o pai terminar (terminação em cascata – *Cascading Termination);*

Se um processo foi encerrado e nenhum pai está em espera, o processo encerrado é conhecido como **processo *zombie***.

- Somente quando o pai chama *wait()*, o ***pid*** do **processo *zombie*** e a sua entrada na tabela de processos são libertados;

Se um processo pai termina antes dos seus filhos, esses filhos, se houver, são conhecidos como **processos órfãos.**

- UNIX/Linux endereça **processos órfãos** atribuindo o processo ***init*** como o novo pai para os **processos órfãos**;
- O processo ***init*** invoca periodicamente *wait()*, permitindo assim que o identificador do processo do órfão e a entrada da tabela de processos sejam libertados.

Os processos dentro de um sistema podem ser **independentes** ou **cooperativos:**

- **Processo independente** não pode afetar ou ser afetado pela execução de outros;
- **Processo cooperativo** pode afetar ou ser afetado pela execução de outros;

Principais razões para **processos cooperativos**:

- **Partilha de informações** – acesso simultâneo à mesma informação;
- **Modularidade** – quebra a computação em subtarefas que fazem mais sentido;
- **Aceleração** – executa subtarefas simultâneas em paralelo;

Para trocar dados e informações, os **processos cooperativos** precisam de suporte para mecanismos de comunicação entre processos (***Interprocess communication – IPC***). Há dois modelos IPC:

- Passagem de mensagens (***Message passing***);
  - Comunicação via envio/receção de mensagens;
  - Funciona em rede;
- Memória partilhada (***Sharing memory***);
  - A comunicação ocorre simplesmente lendo/escrevendo na memória partilhada;
  - Pode levar a problemas complexos de sincronização;

Memória partilhada é mais rápida que a passagem de mensagens

- A passagem de mensagens é implementada usando *system calls* e por isso requerem uma tarefa mais demorada de intervenção do *kernel;*
- Com a memória partilhada, *system calls* só são necessárias para estabelecer as regiões de memória partilhada e, após isso, todos os acessos são tratados como acessos de memória de rotina e com nenhuma intervenção do *kernel*.

Pipes(|) foram um dos primeiros mecanismo ***IPC*** nos primeiros sistemas UNIX e fornecem uma das maneiras mais simples para os processos se comunicarem. Permitem a comunicação em um estilo padrão *producer-consumer:*

- O produtor grava em uma extremidade do canal;
- O consumidor lê do outro lado;

São unidirecionais, permitindo apenas comunicação unidirecional, caso a comunicação bidirecional seja necessárias, dois *pipes* devem ser usados, com cada *pipe* enviando dados em uma direção diferente.

No UNIX/Linux, *pipes* são criadas com o comando *pipe()* do *system call*. O UNIX/Linux trata as *pipes* como um tipo especial de ficheiro, o que permite serem acessadas usando *read()* e *write()* do *system call*.

Pipes não podem ser acessadas de fora do processo onde foram criadas.

- Normalmente, um processo pai cria um ***pipe*** e o usa para se comunicar com um processo filho criado por meio de *fork()*;
- Como um ***pipe*** é um tipo especial de ficheiro, o filho herda o ***pipe*** do processo pai;
- Depois do *fork*, podemos decidir em que direção flui a informação.

Sockets são definidos como um terminal para a comunicação. Um par de comunicação entre processos através de uma rede usa um par de ***sockets*** – um para cada processo. Um ***socket*** é identificado por um endereço IP concatenado com um número. Em geral, os ***socket*** usam uma arquitetura **cliente-servidor**. O servidor aguarda os pedidos recebidos do cliente ouvindo uma porta

especificada. Uma vez que um pedido é recebido, o servidor aceita uma conexão do ***socket*** do cliente para completar a conexão.

IPC Socket é um terminal de comunicação de dados para troca de dados entre processos em execução no mesmo sistema operativo *host*. ***INET Socket*** no Linux são ***sockets*** de *Internet,* ou seja, ***sockets*** baseados no protocolo IP.

Gestão do tempo de CPU

→ Execução por *bursts* CPU e I/O;

→ Processo *scheduler*;

→ Algoritmos com e sem interrupção;
- → *Preemptive;*
- → *Non-preemptive;*

→ Implicações dos algoritmos *preemptive* no desenho do *hardware* e *software*;
- → Necessidade de *timers;*
- → Gestão de "*context switches"*;
- → Inibição de *interrupts* em zonas críticas de código;

→ Métricas de performance;

→ Exemplos de algoritmos e avaliação de métricas, dada uma configuração de processos inicial;
- → Saber calcular sequência de processos;
- → Saber calcular "*Average waiting time*";

→ Algoritmos
- → "*First Come First Served*";
- → "*Round Robin*";
- → "*Shortest Job First*";
- → "*Shortest Remaining Time First*";

→ *Scheduling* com prioridades;

→ *Multi-level queues (MLQ)*;

→ Algoritmo usado no Linux: *Completely Fair Scheduler (CFS)*;

→ Outros tipos de *scheduler* (mencionar apenas)
- → Para sistemas de tempo real;
- → *Earliest Deadline First (EDF)*;
- → *Rate Monotonic Scheduling (RMS)*;

A execução de um processo pode parecer um ciclo de execução da CPU e com tempos de espera de I/O. A execução do processo começa com um ***burst*** de CPU que é seguida por um ***burst*** de I/O, que é seguida por outro ***burst*** de CPU, depois outro ***burst*** de I/O e assim por diante… Eventualmente, o ***burst*** final da CPU termina com um pedido do sistema para terminar a execução.

As decisões de agendamento podem ocorrer quando um processo:

- Muda do estado "em espera" para "pronto" – resultante da conclusão de I/O;
- Muda do estado "em execução" para "pronto" – resultante de uma interrupção;
- Termina;

O ***scheduler*** seleciona entre os processos na fila de "pronto" e aloca a CPU para um deles

- Quando as decisões de agendamento ocorrem nas circunstâncias 1,2 e 4, dizemos que o ***scheduler*** é ***non-preemptive*** (ou cooperativo);
- Caso contrário, o ***scheduler*** é ***preemptive***;

O **agendamento *preemptive*** requer *hardware* especial, como um *timer*.

O **agendamento *preemptive*** pode resultar em condições de corrida (isto é, a saída depende da sequência de execução de outros eventos incontroláveis).

- Enquanto um processo está atualizando dados, ele é antecipado para que um segundo processo possa ser executado. O segundo processo tenta ler os mesmos dados, que podem estar em um estado inconsistente.
- O processamento de uma *system call* pode envolver a alteração de dados importantes do *kernel* (por exemplo, filas de I/O). Se o processo for interrompido no meio dessas alterações e o *kernel* precisar de ler ou modificar a mesma estrutura, ocorrerá caos.

Como as instruções podem ocorrer a qualquer momento, essas secções de código devem ser protegidas de acessos simultâneos por vários processos e, para isso, as **interrupções** são desativadas ao entrar em tais secções e reativadas apenas na saída.

Critérios para comparar algoritmos de agendamento:

- **Utilização da CPU** – mantém a CPU o mais ocupada possível;
- **Taxa de transferência** – número de processos que completam a execução por unidade de tempo;
- **Tempo de retorno/conclusão** – quantidade de tempo necessária para executar um processo;
- **Tempo de espera** – quantidade de tempo que um processo está esperando na fila de "pronto";
- **Tempo de resposta** – quantidade de tempo que leva desde o envio de um pedido até que uma primeira resposta (não *output*) seja produzida;

Critérios de otimização:

- Maximiza a utilização e a taxa de transferência da CPU;
- Minimiza o tempo de resposta, o tempo de espera e o tempo de retorno;

First-Come First-Served (FCFS)

O processo que pede primeiro a CPU, recebe a CPU primeiro.

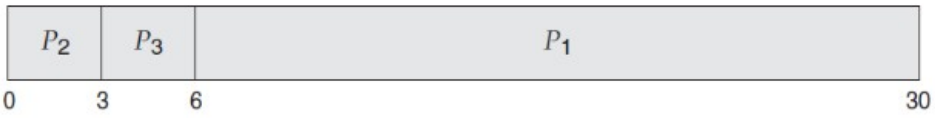
- Gerido por uma fila FIFO;
- Quando um processo entra na fila de "pronto", ele é posto no fim da fila;
- Quando a CPU está livre, ela é alocada ao processo no início da fila de "pronto" (e esse processo é removido da fila);

FCFS é *non-preemptive*, uma vez que a CPU tenha sido alocada a um processo, esse processo mantém a CPU até que termine ou que peça I/O.

- O tempo médio de retorno e espera costuma ser longo;
- Problemático para sistemas de compartilhamento de tempo, onde é importante que cada utilizador obtenha uma parte da CPU em intervalos regulares;

Process	Burst Time
P_1	24
P_2	3
P_3	3

- FCFS average waiting time
  - (0 + 24 + 27) / 3 = 51 / 3 = 17



Process	Burst Time
P_2	3
P_3	3
P_1	24

- FCFS average waiting time
  - (0 + 3 + 6) / 3 = 9 / 3 = 3

Round Robin (RR)

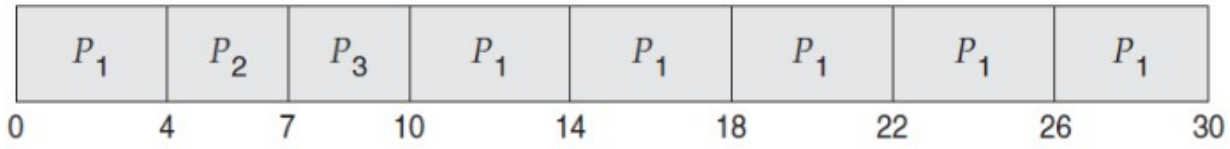
Tipo de ***FCFS*** mas *preemptive*, especialmente desenhado para sistemas de compartilhamento de tempo.

- Cada processo recebe um *quantum* de tempo (pequena unidade de tempo da CPU);
- O temporizador interrompe cada *quantum* para agendar o próximo processo, o processo atual é antecipado e adicionado ao final da fila de "pronto" (a fila de "pronto" funciona como uma fila circular);

Se o *quantum* de tempo for ***Q*** e houver ***N*** processos na fila de "pronto", cada processo obterá 1/***N*** do tempo de CPU em pedaços de no máximo ***Q*** unidades de tempo de uma só vez (nenhum processo espera mais que ***(N-1)*Q*** unidades de tempo).

- ***Q*** grande → igual ao ***FCFS***;
- ***Q*** pequeno → aumenta o número de *context switches*, a sobrecarga pode ser muito alta.

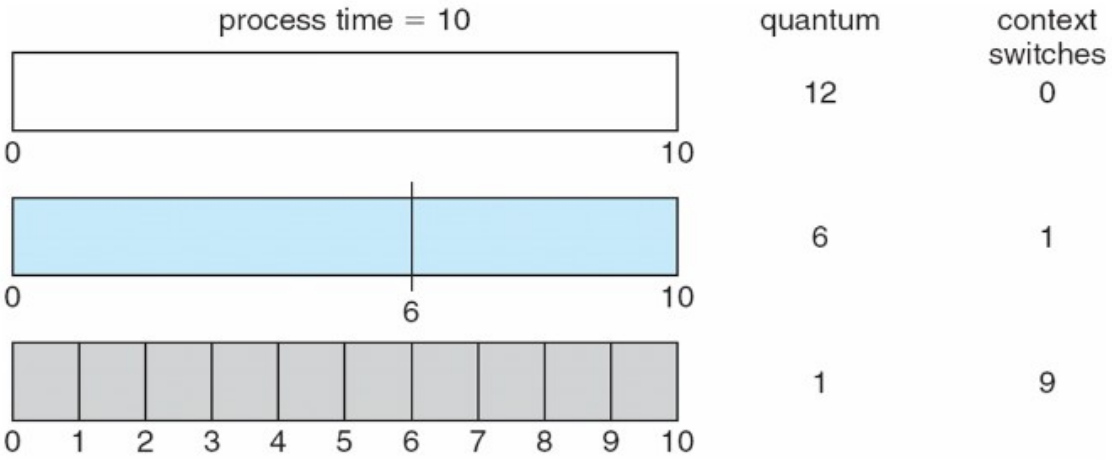
Process	Burst Time
P_1	24
P_2	3
P_3	3



RR (time quantum 4) average waiting time

- (6 + 4 + 7) / 3 = 17 / 3 = 5.66

Time Quantum x Context Switch Time



Shortest-Job-First (SJF)

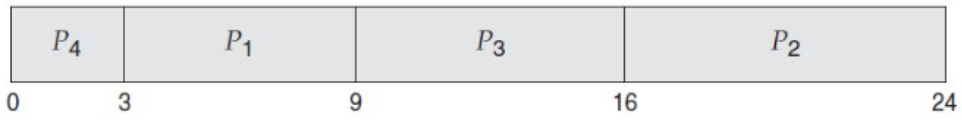
Associa cada processo com a duração do seu próximo *burst* de CPU e usa esses comprimentos para agendar o processo com o *burst* de CPU mais baixo.

- Se os próximos *bursts* de CPU de dois processos forem os mesmos, o agendamento para desempate é o ***FCFS***;

SJF é ideal porque sempre fornece o tempo médio de espera mínimo para um determinado conjunto de processos.

- Mover um processo curto antes de um longo diminui o tempo de espera do processo curto mais do que aumenta o tempo de espera do processo longo;
- A dificuldade é saber a duração do próximo *burst* da CPU.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



SJF average waiting time

- (0 + 3 + 9 + 16) / 4 = 28 / 4 = 7

FCFS average waiting time

- (0 + 6 + 14 + 21) / 4 = 41 / 4 = 10.25

Shortest-Remaining-Time-First (SRTF)

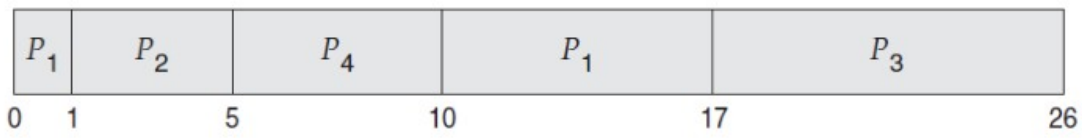
SJF pode ser *non-preemptive* ou *preemptive*.

- O agendamento ***SJF*** *preemptive* é chamado de ***SRFT***;

A escolha de ser *preemptive* ou não ocorre quando um novo processo chega à fila de "pronto" e o próximo *burst* de CPU do processo recém-chegado pode ser menor do que o que resta do processo atualmente em execução.

- O agendamento ***SJF*** (*non-preemptive)* permitirá que o processo atualmente em execução termine o *burst* de CPU;
- O agendamento ***SRTF*** (*preemptive*) irá antecipar o processo atualmente em execução e agendar o processo recém-chegado;

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



SRTF (preemptive) average waiting time

- [(17-8-0) + (5-4-1) + (26-9-2) + (10-5-3)] / 4 = 26 / 4 = 6.5

SJF (nonpreemptive) average waiting time

- [0 + (8-1) + (12-3) + (17-2)] / 4 = 31 / 4 = 7.75

O agendamento por prioridade (***Priority scheduling***) associa um número de prioridade a cada processo e a CPU é alocada ao processo de maior prioridade.

- ***SJF*** e ***SRTF*** podem ser vistos como algoritmos de prioridade;

O **agendamento prioritário** pode ser:

- *Preemptive*, antecipa a CPU se a prioridade do processo recém-chegado for maior que a prioridade do processo em execução;
- *Non-preemptive*, permite que o processo em execução finalize o seu *burst* de CPU;

Um grande problema é o bloqueio indefinido/"fome".

- Processos de baixa prioridade podem nunca ser executados e esperar indefinidamente;
- Uma solução comum é o envelhecimento, o que envolve aumentar gradualmente a prioridade de processos que aguardam no sistema por muito tempo.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Priority scheduling average waiting time

- (0 + 1 + 6 + 16 + 18) / 5 = 41 / 5 = 8.2

FCFS

- **(+)** simples;
- **(-)** processos curtos ficam presos atrás dos longos;

RR

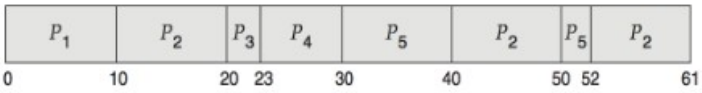
- **(+)** melhor para processos mais curtos;
- **(-)** o tempo de *context switch* aumenta para trabalhos longos;

SJF e SRTF

- **(+)** tempo de espera médio ideal;
- **(+)** melhor efeito em processos curtos;
- **(-)** difícil prever o futuro;
- **(-)** bloqueio indefinido/"fome"

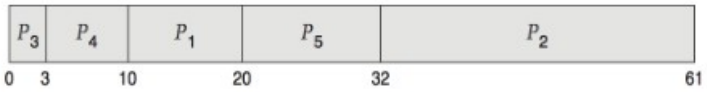
FCFS X RR x SJF: One Last Example

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



FCFS average waiting time

- (0 + 10 + 39 + 42 + 49) / 5 = 28



RR (time quantum 10) average waiting time

- [0 + (61-29) + 20 + 23 + (52-12)] / 5 = 115 / 5 = 23

SJF (nonpreemptive) average waiting time

- (0 + 3 + 10 + 20 + 32) / 5 = 65 / 5 = 13

O agendamento ***MLQ (Multi-level queue)*** divide a fila "pronto" em várias filas separadas.

- Os processos são atribuídos permanentemente a uma fila, geralmente com base em alguma propriedade do processo, como tamanho da memória, prioridade do processo, …;
- Cada fila tem seu próprio algoritmo de agendamento;

Além disso, deve haver agendamento entre as filas:

- **Agendamento de prioridade fixa** – cada fila tem prioridade absoluta sobre as filas de prioridade mais baixa (agendamento *preemptive* com possibilidade de bloqueios indefinidos);
- **Intervalo de tempo** – cada fila recebe uma certa quantidade de tempo de CPU que é então agendado entre os seus processos;

O agendamento ***MLQ*** envolve a definição de quatro parâmetros:

- Número de filas;
- Algoritmo de agendamento para cada fila;
- Algoritmo de agendamento entre as filas (prioridade fixa ou intervalo de tempo);
- Método para determinar a qual fila um processo será atribuído;

Prós e contras:

- **(+)** baixa sobrecarga de agendamento;
- **(-)** o agendamento de prioridade fixa é injusto, inflexível e pode levar à "fome"/bloqueios indefinidos;
- **(-)** o intervalo de tempo pode prejudicar o tempo médio de espera;

CFS (***Completely Fair Scheduler***) é o algoritmo de agendamento adotado pelo *kernel* do Linux. O ***CFS*** tenta dividir o tempo da CPU de forma justa entre todas as tarefas (processos ou *threads*) levando em consideração as suas prioridades e histórico de uso da CPU. É baseado em classes de agendamento em que cada classe tem um intervalo de prioridade específico.

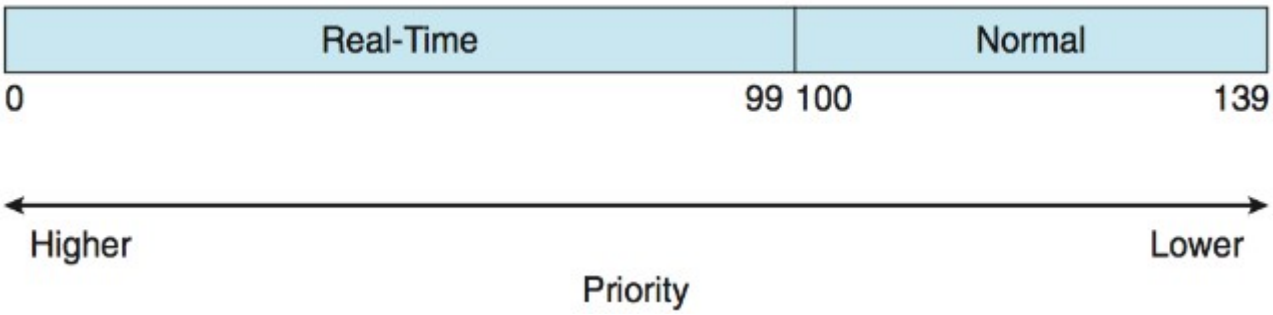
- O agendador seleciona a tarefa de prioridade mais elevada da classe de prioridade mais elevada.
- As tarefas de prioridade mais baixa são antecipadas quando as tarefas de prioridade mais alta estão prontas para execução.

Normalmente, o *kernel* do Linux implementa duas classes de agendamento:

- Classe em tempo real;
- Classe normal (padrão);

A classe em tempo real mais a classe normal mapeiam em um intervalo de prioridade global:

- As tarefas em tempo real recebem prioridades estáticas dentro do intervalo [0,99];
- Tarefas normais têm bons valores e são atribuídas prioridade dinâmicas dentro do intervalo [100,139] com base nos seus bons valores;
- Os valores bons variam de [-20,+19] e mapeiam para prioridades [100,139] (o valor padrão de bom é 0);
- Um valor agradável mais baixo/mais alto significa prioridade mais alta/baixa (a ideia é que, se uma tarefa aumenta o seu valor, ela está sendo boa para as outras tarefas no sistema);



As tarefas em tempo real são agendadas por prioridade e antes das tarefas em outras classes. As tarefas normais são agendadas de acordo com o menor valor de tempo de execução virtual.

- O ***CFS*** mantém um valor de tempo de execução virtual por tarefa que mede o tempo de CPU associando um fator de decaimento com base no valor bom da tarefa.
- Valores bons de 0 geram um tempo de execução virtual idêntico ao tempo de execução real (se uma tarefa for executada por 100 milissegundos, seu tempo de execução virtual também será de 100 milissegundos).
- Tarefas de prioridade mais baixa têm fatores de decaimento mais altos, enquanto que tarefas de prioridade mais alta têm fatores de atraso mais baixos (se uma tarefa for executada por 100 milissegundos, seu tempo de execução virtual será proporcionalmente maior/menor que 100 milissegundos de acordo com sua prioridade mais baixa/mais alta).
- Quando uma nova tarefa é criada, é atribuído a ela um tempo de execução virtual igual ao tempo de execução virtual mínimo atual.

Com o ***CFS***, as tarefas não têm intervalos de tempo fixas, mas são executadas até que não sejam mais as tarefas mais injustamente tratadas. O ***CFS*** identifica uma latência de objetivo, que é

um intervalo de tempo durante o qual cada tarefa executável deve ser executada pelo menos uma vez.

- A latência de objetivo tem valores padrão e mínimo, mas pode aumentar se o número de tarefas ativas no sistema ultrapassar um determinado limite

As tarefas obtêm proporções de tempo de CPU do valor de latência de objetivo de acordo com as suas prioridades relativas. Quando uma tarefa é despertada, a diferença de seu tempo de execução virtual para o tempo de execução mínimo atual não pode exceder a latência de objetivo, caso contrário, seu tempo de execução virtual é ajustado para esse limite.

- Isso evita que uma tarefa que está esperando muito tempo monopolize a CPU.

Gestão da Memória Física

→ A hierarquia de memória
  → CPU;
  → *Caches*;
  → Memória física;
  → Disco;

→ Espaço de endereçamento de um processo;

→ Endereços virtuais e físicos;

→ Limites do espaço de endereçamento
  → Em arquiteturas 32 bits;
  → Em arquiteturas 64 bits;

→ Mapeamento e proteção de memória;

→ *Memory Management Unit* (*MMU*);

→ Técnicas de gestão contígua de memória;

→ Fragmentação interna e externa;

→ *Swapping*
  → Vantagens;
  → Desvantagens;

→ Técnicas de gestão não contíguas de memória.

→ Segmentação
  → Noção de segmento;
  → Tabela de segmentos;
  → Tradução de endereços virtuais;
  → Proteção da memória;